Laravel 9 · Валидация

Введение

Laravel предлагает несколько подходов для проверки входящих данных вашего приложения. Например, метод `validate`, доступен для всех входящих HTTP-запросов. Однако мы обсудим и другие подходы к валидации.

Laravel содержит удобные правила валидации, применяемые к данным, включая валидацию на уникальность значения в конкретной таблице базы данных. Мы подробно рассмотрим каждое из этих правил валидации, чтобы вы были знакомы со всеми особенностями валидации Laravel.

Быстрый старт

Чтобы узнать о мощных функциях валидации Laravel, давайте рассмотрим полный пример валидации формы и отображения сообщений об ошибках конечному пользователю. Прочитав этот общий обзор, вы сможете получить представление о том, как проверять данные входящего запроса с помощью Laravel:

Определение маршрутов

Во-первых, предположим, что в нашем файле `routes/web.php` определены следующие маршруты:

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

Маршрут `GET` отобразит форму для пользователя для создания нового сообщения в блоге, а маршрут `POST` сохранит новое сообщение в базе данных.

Создание контроллера

Затем, давайте взглянем на простой контроллер, который обрабатывает входящие запросы на эти маршруты. Пока оставим метод `store` пустым:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Показать форму для создания нового сообщения в блоге.
     *
     * @return \Illuminate\View\View
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Сохранить новую запись в блоге.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Выполнить валидацию и сохранить сообщение в блоге ...
    }
}
```

Написание логики валидации

Теперь мы готовы заполнить наш метод `store` логикой для валидации нового сообщения в блоге. Для этого мы будем использовать метод `validate`, предоставляемый объектом `Illuminate\Http\Request`. Если правила валидации будут пройдены, то ваш код продолжит нормально выполняться; однако, если проверка не пройдена, то будет выброшено исключение `Illuminate\Validation\ValidationException`, и соответствующий ответ об ошибке будет автоматически отправлен обратно пользователю.

Если валидации не пройдена во время традиционного HTTP-запроса, то будет сгенерирован ответ-перенаправление на предыдущий URL-адрес. Если входящий запрос является XHR-запросом, то будет возвращен JSON-ответ, содержащий сообщения об ошибках валидации.

Чтобы лучше понять метод `validate`, давайте вернемся к методу `store`:

```
/**
 * Сохранить новую запись в блоге.
```

```
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $validated = $request->validate([
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        // Запись блога корректна ...
    }
```

Как видите, правила валидации передаются в метод `validate`. Не волнуйтесь – все доступные правила валидации задокументированы. Опять же, если проверка не пройдена, то будет автоматически сгенерирован корректный ответ. Если проверка пройдет успешно, то наш контроллер продолжит нормальную работу.

В качестве альтернативы правила валидации могут быть указаны как массивы правил вместо одной строки с разделителями `|`:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Кроме того, вы можете использовать метод `validateWithBag` для валидации запроса и сохранения любых сообщений об ошибках в именованную коллекцию ошибок:

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Прекращение валидации при возникновении первой ошибки

По желанию можно прекратить выполнение правил валидации для атрибута после первой ошибки. Для этого присвойте атрибуту правило `bail`:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

В этом примере, если правило `unique` для атрибута `title` не будет пройдено, то правило `max` не будет выполняться. Правила будут проверяться в порядке их назначения.

Примечание о вложенных атрибутах

Если входящий HTTP-запрос содержит данные «вложенных» полей, то вы можете указать эти поля в своих правилах валидации, используя «точечную нотацию»:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

С другой стороны, если имя вашего поля буквально содержит точку, то вы можете явно запретить ее интерпретацию как часть «точечной нотации», экранировав точку с помощью обратной косой черты:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'v1\.0' => 'required',
]);
```

Отображение ошибок валидации

Итак, что, если поля входящего запроса не проходят указанные правила валидации? Как упоминалось ранее, Laravel автоматически перенаправит пользователя обратно в его предыдущее местоположение. Кроме того, все ошибки валидации и входящие данные запроса будут автоматически записаны в сессию.

Переменная `$errors` используется во всех шаблонах вашего приложения благодаря посреднику `Illuminate\View\Middleware\ShareErrorsFromSession`, который включен в группу посредников `web`. Пока применяется этот посредник, в ваших шаблонах всегда будет доступна переменная `$errors`, что позволяет вам предполагать, что переменная `$errors` всегда определена и может безопасно использоваться. Переменная `$errors` будет экземпляром `Illuminate\Support\MessageBag`. Для получения дополнительной информации о работе с этим объектом ознакомьтесь с его документацией.

Итак, в нашем примере пользователь будет перенаправлен на метод нашего контроллера `create`, в случае, если валидация завершится неудачно, что позволит нам отобразить сообщения об ошибках в шаблоне:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Создание поста блога</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
```

```
            @endforeach
        </ul>
    </div>
@endif

<!-- Форма для создания поста блога -->
```

Корректировка сообщений об ошибках

Каждое встроенное правило валидации Laravel содержит сообщение об ошибке, которое находится в файле `lang/en/validation.php` вашего приложения. В этом файле вы найдете запись о переводе для каждого правила валидации. Вы можете изменять или модифицировать эти сообщения в зависимости от потребностей вашего приложения.

Кроме того, вы можете скопировать этот файл в каталог перевода другого языка, чтобы перевести сообщения на язык вашего приложения. Чтобы узнать больше о локализации Laravel, ознакомьтесь с полной документацией по локализации.

XHR-запросы и валидация

В этом примере мы использовали традиционную форму для отправки данных в приложение. Однако, многие приложения получают запросы XHR с фронтенда с использованием JavaScript. При использовании метода `validate`, во время выполнения XHR-запроса, Laravel не будет генерировать ответ-перенаправление. Вместо этого Laravel генерирует JSON-ответ, содержащий все ошибки валидации. Этот ответ JSON будет отправлен с кодом 422 состояния HTTP.

Директива `@error`

Вы можете использовать директиву `@error` Blade, чтобы быстро определить, существуют ли сообщения об ошибках валидации для конкретного атрибута, включая сообщения об ошибках в именованной коллекции ошибок. В директиве `@error` вы можете вывести содержимое переменной `$message` для отображения сообщения об ошибке:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
    type="text"
    name="title"
    class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Если вы используете именованные коллекции ошибок, то вы можете передать имя коллекции в качестве второго аргумента директивы `@error`:

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

Повторное заполнение форм

Когда Laravel генерирует ответ-перенаправление из-за ошибки валидации, фреймворк автоматически краткосрочно записывает все входные данные запроса в сессию. Это сделано для того, чтобы вы могли удобно получить доступ к входным данным во время следующего запроса и повторно заполнить форму, которую пользователь попытался отправить.

Чтобы получить входные данные предыдущего запроса, вызовите метод `old` экземпляра `Illuminate\Http\Request`. Метод `old` извлечет ранее записанные входные данные из сессии:

```
$title = $request->old('title');
```

Laravel также содержит глобального помощника `old`. Если вы показываете входные данные прошлого запроса в шаблоне Blade, то удобнее использовать помощник `old` для повторного заполнения формы. Если для какого-то поля не были предоставлены данные в прошлом запросе, то будет возвращен `null`:

```
<input type="text" name="title" value="{{ old('title') }}">
```

Примечание о необязательных полях

По умолчанию Laravel содержит посредников `App\Http\Middleware\TrimStrings` и `App\Http\Middleware\ConvertEmptyStringsToNull` в глобальном стеке посредников вашего приложения. Эти посредники перечислены в классе `App\Http\Kernel`. Первый из упомянутых посредников будет автоматически обрезать все входящие строковые поля запроса, а второй – конвертировать любые пустые строковые поля в `null`. Из-за этого вам часто нужно будет помечать ваши «необязательные» поля запроса как `nullable`, если вы не хотите, чтобы валидатор не считал такие поля недействительными. Например:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

В этом примере мы указываем, что поле `publish_at` может быть либо `null`, либо допустимым представлением даты. Если модификатор `nullable` не добавлен в определение правила, валидатор сочтет `null` недопустимой датой.

Формат ответа ошибок валидации

Когда ваше приложение выбрасывает исключение `Illuminate\Validation\ValidationException` и на входящий HTTP-запрос ожидается ответ JSON, то Laravel автоматически отформатирует для вас сообщения об ошибках и вернет HTTP-ответ `422 Unprocessable Entity`.

Ниже вы можете просмотреть пример формата ответа JSON для ошибок валидации. Обратите внимание, что вложенные ключи ошибок сведены к «точечной» нотации:

```
{
    "message": "The team name must be a string. (and 4 more errors)",
    "errors": {
        "team_name": [
            "The team name must be a string.",
            "The team name must be at least 1 characters."
        ],
        "authorization.role": [
            "The selected authorization.role is invalid."
        ],
        "users.0.email": [
            "The users.0.email field is required."
        ],
        "users.2.email": [
            "The users.2.email must be a valid email address."
        ]
    }
}
```

Валидация запроса формы

Создание запросов формы

Для более сложных сценариев валидации вы можете создать «запрос формы». Запрос формы – это ваш класс запроса, который инкапсулирует свою собственную логику валидации и авторизации. Чтобы сгенерировать новый запрос формы, используйте команду `make:request` Artisan:

```
php artisan make:request StorePostRequest
```

Эта команда поместит новый класс запроса формы в каталог `app/Http/Requests` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его, когда вы запустите команду `make:request`. Каждый запрос формы, созданный Laravel, имеет два метода: `authorize` и `rules`.

Как вы могли догадаться, метод `authorize` отвечает за определение того, может ли текущий аутентифицированный пользователь выполнить действие, представленное запросом, в то время как метод `rules` возвращает правила валидации, которые должны применяться к данным запроса:

```
/**
 * Получить массив правил валидации, которые будут применены к запросу.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

> {tip} Вы можете объявить любые зависимости, которые вам нужны, в сигнатуре метода `rules` . Они будут автоматически извлечены через контейнер служб Laravel.

Итак, как анализируются правила валидации? Все, что вам нужно сделать, это объявить зависимость от запроса в методе вашего контроллера. Входящий запрос формы проверяется до вызова метода контроллера, что означает, что вам не нужно загромождать контроллер какой-либо логикой валидации:

```
/**
 * Сохранить новую запись в блоге.
 *
 * @param  \App\Http\Requests\StorePostRequest  $request
 * @return Illuminate\Http\Response
 */
public function store(StorePostRequest $request)
{
    // Входящий запрос прошел валидацию ...

    // Получить провалидированные входные данные ...
    $validated = $request->validated();

    // Получить часть провалидированных входных данных ...
    $validated = $request->safe()->only(['name', 'email']);
    $validated = $request->safe()->except(['name', 'email']);
}
```

При неуспешной валидации будет сгенерирован ответ-перенаправление, чтобы отправить пользователя обратно в его предыдущее местоположение. Ошибки также будут краткосрочно записаны в сессию, чтобы они были доступны для отображения. Если запрос был XHR-запросом, то пользователю будет возвращен HTTP-ответ с кодом состояния 422, включая JSON-представление ошибок валидации.

Добавление хуков `after` для запросов форм

Если вы хотите добавить хук валидации `after` для запроса формы, то вы можете использовать метод `withValidator`. Этот метод получает полностью инициированный валидатор, что позволяет вам вызвать любой из его методов до того, как правила валидации будут фактически проанализированы:

```
/**
 * Надстройка экземпляра валидатора.
 *
 * @param  \Illuminate\Validation\Validator  $validator
 * @return void
 */
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Что-то не так с этим полем!');
        }
    });
}
```

Прекращение валидации после первой неуспешной проверки

Добавив свойство `$stopOnFirstFailure` вашему классу запроса, вы можете сообщить валидатору, что он должен прекратить валидацию всех атрибутов после возникновения первой ошибки валидации:

```
/**
 * Остановить валидацию после первой неуспешной проверки.
 *
 * @var bool
 */
protected $stopOnFirstFailure = true;
```

Изменение адреса ответа-перенаправления

Как обсуждалось ранее, будет сгенерирован ответ-перенаправление, чтобы отправить пользователя обратно в его предыдущее местоположение, если валидация запроса формы является неуспешной. Однако вы можете настроить это поведение. Для этого определите свойство `$redirect` в вашем запросе формы:

```
/**
 * URI перенаправления пользователей в случае неуспешной валидации.
 *
 * @var string
 */
protected $redirect = '/dashboard';
```

Или, если вы хотите перенаправить пользователей на именованный маршрут, то вы можете вместо этого определить свойство `$redirectRoute`:

```
/**
 * Маршрут перенаправления пользователей в случае неуспешной валидации.
 *
 * @var string
 */
protected $redirectRoute = 'dashboard';
```

Авторизация запросов

Класс запроса формы также содержит метод `authorize`. В рамках этого метода вы можете определить, действительно ли аутентифицированный пользователь имеет право изменять текущий ресурс. Например, вы можете определить, действительно ли пользователь владеет комментарием в блоге, который он пытается обновить. Скорее всего, вы будете взаимодействовать с вашими шлюзами и политиками авторизации в этом методе:

```
use App\Models\Comment;

/**
 * Определить, уполномочен ли пользователь выполнить этот запрос.
 *
 * @return bool
 */
public function authorize()
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

Поскольку все запросы формы расширяют базовый класс запросов Laravel, мы можем использовать метод `user` для доступа к текущему аутентифицированному пользователю. Также обратите внимание на вызов метода `route` в приведенном выше примере. Этот метод обеспечивает вам доступ к параметрам URI, определенным для вызываемого маршрута, таким как параметр `{comment}` в приведенном ниже примере:

```
Route::post('/comment/{comment}');
```

Если ваше приложение использует привязку модели к маршруту, то ваш код можно сделать еще более лаконичным, если обратиться к извлеченной модели как к свойству текущего запроса:

```
return $this->user()->can('update', $this->comment);
```

Если метод `authorize` возвращает `false`, то будет автоматически возвращен HTTP-ответ с кодом состояния 403, и метод вашего контроллера не будет выполнен.

Если вы планируете обрабатывать логику авторизации для запроса в другой части вашего приложения, то вы можете просто вернуть `true` из метода `authorize`:

```
/**
 * Определить, уполномочен ли пользователь выполнить этот запрос.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

> {tip} Вы можете объявить любые зависимости, которые вам нужны, в сигнатуре метода `authorize`. Они будут автоматически извлечены через контейнер служб Laravel.

Корректировка сообщений об ошибках

Вы можете изменить сообщения об ошибках, используемые в запросе формы, переопределив метод `messages`. Этот метод должен возвращать массив пар атрибут / правило и соответствующие им сообщения об ошибках:

```
/**
 * Получить сообщения об ошибках для определенных правил валидации.
 *
 * @return array
 */
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'body.required' => 'A message is required',
    ];
}
```

Корректировка атрибутов валидации

Многие сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:attribute`. Если вы хотите, чтобы заполнитель `:attribute` вашего сообщения валидации был заменен другим именем атрибута, то вы можете указать собственные имена, переопределив метод `attributes`. Этот метод должен возвращать массив пар атрибут / имя:

```
/**
 * Получить пользовательские имена атрибутов для формирования ошибок валидатора.
 *
 * @return array
```

```
     */
    public function attributes()
    {
        return [
            'email' => 'email address',
        ];
    }
```

Подготовка входящих данных для валидации

Если вам необходимо подготовить или обработать какие-либо данные из запроса перед применением правил валидации, то вы можете использовать метод `prepareForValidation`:

```
use Illuminate\Support\Str;

/**
 * Подготовить данные для валидации.
 *
 * @return void
 */
protected function prepareForValidation()
{
    $this->merge([
        'slug' => Str::slug($this->slug),
    ]);
}
```

Создание валидатора по требованию

Если вы не хотите использовать метод `validate` запроса, то вы можете создать экземпляр валидатора вручную, используя фасад `Validator`. Метод `make` фасада генерирует новый экземпляр валидатора:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * Сохранить новую запись в блоге.
     *
     * @param  Request  $request
     * @return Response
     */
```

```
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Получить провалидированные входные данные ...
        $validated = $validator->validated();

        // Получить часть провалидированных входных данных ...
        $validated = $validator->safe()->only(['name', 'email']);
        $validated = $validator->safe()->except(['name', 'email']);

        // Сохранить сообщение блога ...
    }
}
```

Первым аргументом, переданным методу `make`, являются проверяемые данные. Второй аргумент – это массив правил валидации, которые должны применяться к данным.

После определения того, что запрос не прошел валидацию с помощью метода `fails`, вы можете использовать метод `withErrors` для передачи сообщений об ошибках в сессию. При использовании этого метода переменная `$errors` будет автоматически передана вашим шаблонам после перенаправления, что позволит вам легко отобразить их обратно пользователю. Метод `withErrors` принимает экземпляр валидатора, экземпляр `MessageBag` или обычный массив PHP.

Прекращение валидации после первой неуспешной проверки

Метод `stopOnFirstFailure` проинформирует валидатор о том, что он должен прекратить валидацию всех атрибутов после возникновения первой ошибки валидации:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

Автоматическое перенаправление

Если вы хотите создать экземпляр валидатора вручную, но по-прежнему воспользоваться преимуществами автоматического перенаправления, предлагаемого методом `validate` HTTP-запроса, вы можете вызвать метод `validate` созданного экземпляра валидатора. Пользователь будет автоматически перенаправлен или, в случае запроса XHR, будет возвращен ответ JSON, если валидация будет не успешной:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Вы можете использовать метод `validateWithBag` для сохранения сообщений об ошибках в именованной коллекции ошибок, если валидация будет не успешной:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validateWithBag('post');
```

Именованные коллекции ошибок

Если у вас есть несколько форм на одной странице, то вы можете задать имя экземпляру `MessageBag`, содержащий ошибки валидации, что позволит вам получать сообщения об ошибках для конкретной формы. Чтобы добиться этого, передайте имя в качестве второго аргумента в метод `withErrors`:

```
return redirect('register')->withErrors($validator, 'login');
```

Затем, вы можете получить доступ к именованному экземпляру `MessageBag` из переменной `$errors`:

```
{{ $errors->login->first('email') }}
```

Корректировка сообщений об ошибках

При необходимости вы можете предоставить собственные сообщения об ошибках, которые должен использовать экземпляр валидатора вместо сообщений об ошибках по умолчанию, предоставляемых Laravel. Есть несколько способов указать собственные сообщения. Во-первых, вы можете передать собственные сообщения в качестве третьего аргумента методу `Validator::make`:

```
$validator = Validator::make($input, $rules, $messages = [
    'required' => 'The :attribute field is required.',
]);
```

В этом примере заполнитель `:attribute` будет заменен фактическим именем проверяемого поля. Вы также можете использовать другие заполнители в сообщениях валидатора. Например:

```
$messages = [
    'same' => 'The :attribute and :other must match.',
    'size' => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute value :input is not between :min - :max.',
    'in' => 'The :attribute must be one of the following types: :values',
];
```

Указание пользовательского сообщения для конкретного атрибута

По желанию можно указать собственное сообщение об ошибке только для определенного атрибута. Вы можете сделать это, используя «точечную нотацию». Сначала укажите имя атрибута, а затем правило:

```
$messages = [
    'email.required' => 'We need to know your email address!',
];
```

Указание пользовательских имен для атрибутов

Многие сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:attribute`, который заменяется именем проверяемого поля или атрибута. Чтобы указать собственные значения, используемые для замены этих заполнителей для конкретных полей, вы можете передать массив ваших атрибутов в качестве четвертого аргумента методу `Validator::make`:

```
$validator = Validator::make($input, $rules, $messages, [
    'email' => 'email address',
]);
```

Хук валидатора After

Вы также можете определить замыкания, которые будут запускаться после завершения валидации. Это позволяет легко выполнять дальнейшую валидацию и даже добавлять сообщения об ошибках в коллекцию сообщений. Вызовите метод `after` на экземпляре валидатора:

```
$validator = Validator::make(/* ... */);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add(
            'field', 'Something is wrong with this field!'
        );
    }
});
```

```
if ($validator->fails()) {
    //
}
```

Работа с провалидированными входящими данными

После валидации данных входящего запроса с помощью запроса формы или вручную созданного экземпляра валидатора, вы можете получить данные входящего запроса, которые действительно прошли валидацию. Это можно сделать несколькими способами. Во-первых, вы можете вызвать метод `validated` запроса формы или экземпляра валидатора. Этот метод возвращает массив данных, которые были провалидированы:

```
$validated = $request->validated();

$validated = $validator->validated();
```

В качестве альтернативы вы можете вызвать метод `safe` запроса формы или экземпляра валидатора. Этот метод возвращает экземпляр `Illuminate\Support\ValidatedInput`. Этот объект предоставляет методы `only`, `except` и `all` для получения как подмножества, так и целого массива провалидированных данных:

```
$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();
```

Кроме того, экземпляр `Illuminate\Support\ValidatedInput` может быть итерирован и доступен как массив:

```
// Провалидированные данные могут быть итерированы ...
foreach ($request->safe() as $key => $value) {
    //
}

// К провалидированным данным можно получить доступ как к массиву ...
$validated = $request->safe();

$email = $validated['email'];
```

Если вы хотите добавить дополнительные поля к провалидированные данным, то вы можете вызвать метод `merge`:

```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

Если вы хотите получить провалидированные данные как экземпляр коллекции, то вы можете вызвать метод `collect`:

```
$collection = $request->safe()->collect();
```

Работа с сообщениями об ошибках

После вызова метода `errors` экземпляр `Validator`, вы получите экземпляр `Illuminate\Support\MessageBag`, который имеет множество удобных методов для работы с сообщениями об ошибках. Переменная `$errors`, которая автоматически становится доступной для всех шаблонов, также является экземпляром класса `MessageBag`.

Получение первого сообщения об ошибке для поля

Чтобы получить первое сообщение об ошибке для указанного поля, используйте метод `first`:

```
$errors = $validator->errors();

echo $errors->first('email');
```

Получение всех сообщений об ошибках для поля

Если вам нужно получить массив всех сообщений для указанного поля, используйте метод `get`:

```
foreach ($errors->get('email') as $message) {
    //
}
```

Если вы проверяете массив полей формы, то вы можете получить все сообщения для каждого из элементов массива, используя символ `*`:

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

Получение всех сообщений об ошибках для всех полей

Чтобы получить массив всех сообщений для всех полей, используйте метод `all`:

```
foreach ($errors->all() as $message) {
    //
}
```

Определение наличия сообщений для поля

Метод `has` используется для определения наличия сообщений об ошибках для указанного поля:

```
if ($errors->has('email')) {
    //
}
```

Указание пользовательских сообщений в языковых файлах

Каждое встроенное правило валидации Laravel содержит сообщение об ошибке, которое находится в файле `lang/en/validation.php` вашего приложения. В этом файле вы найдете запись о переводе для каждого правила валидации. Вы можете изменять или модифицировать эти сообщения в зависимости от потребностей вашего приложения.

Кроме того, вы можете скопировать этот файл в каталог перевода другого языка, чтобы перевести сообщения на язык вашего приложения. Чтобы узнать больше о локализации Laravel, ознакомьтесь с полной документацией по локализации.

Указание пользовательского сообщения для конкретного атрибута

Вы можете изменить сообщения об ошибках, используемые для указанных комбинаций атрибутов и правил в языковых файлах валидации вашего приложения. Для этого добавьте собственные сообщения в массив `custom` языкового файла `lang/xx/validation.php` вашего приложения:

```
'custom' => [
    'email' => [
        'required' => 'We need to know your email address!',
        'max' => 'Your email address is too long!'
    ],
],
```

Указание атрибутов в языковых файлах

Многие сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:attribute` , который заменяется именем проверяемого поля или атрибута. Если вы хотите, чтобы часть `:attribute` вашего сообщения валидации была заменена собственным значением, то вы можете указать имя настраиваемого атрибута в массиве `attributes` вашего языкового файла `lang/xx/validation.php` :

```
'attributes' => [
    'email' => 'email address',
```

```
    ],
```

Указание пользовательских имен для атрибутов в языковых файлах

Некоторые сообщения об ошибках встроенных правил валидации Laravel содержат заполнитель `:value`, который заменяется текущим значением атрибута запроса. Иногда требуется заменить часть `:value` вашего сообщения валидации на собственное значение. Например, рассмотрим следующее правило, которое указывает, что номер кредитной карты требуется обязательно, если для параметра `payment_type` установлено значение `cc`:

```
Validator::make($request->all(), [
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

Если это правило валидации не будет пройдено, то будет выдано следующее сообщение об ошибке:

```
The credit card number field is required when payment type is cc.
```

Вместо того, чтобы отображать `cc` в качестве значения типа платежа, вы можете указать более удобное для пользователя представление значения в вашем языковом файле `lang/xx/validation.php`, определив массив `values`:

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```

После определения этого значения правило валидации выдаст следующее сообщение об ошибке:

```
The credit card number field is required when payment type is credit card.
```

Доступные правила валидации

Ниже приведен список всех доступных правил валидации и их функций:

- Accepted
- Accepted If
- Active URL
- After (Date)
- After Or Equal (Date)

- Alpha
- Alpha Dash
- Alpha Numeric
- Array
- Bail
- Before (Date)
- Before Or Equal (Date)
- Between
- Boolean
- Confirmed
- Current Password
- Date
- Date Equals
- Date Format
- Declined
- Declined If
- Different
- Digits
- Digits Between
- Dimensions (Image Files)
- Distinct
- Email
- Ends With
- Enum
- Exclude
- Exclude If
- Exclude Unless
- Exclude With
- Exclude Without
- Exists (Database)
- File
- Filled
- Greater Than
- Greater Than Or Equal
- Image (File)
- In
- In Array
- Integer
- IP Address
- JSON

- Less Than
- Less Than Or Equal
- MAC Address
- Max
- MIME Types
- MIME Type By File Extension
- Min
- Multiple Of
- Not In
- Not Regex
- Nullable
- Numeric
- Password
- Present
- Prohibited
- Prohibited If
- Prohibited Unless
- Prohibits
- Regex (regular expression)
- Required
- Required If
- Required Unless
- Required With
- Required With All
- Required Without
- Required Without All
- Required Array Keys
- Same
- Size
- Sometimes
- Starts With
- String
- Timezone
- Unique (Database)
- URL
- UUID

accepted

Проверяемое поле должно иметь значение `"yes"`, `"on"`, `1` или `true`. Применяется для валидации принятия раздела «Условия использования» или аналогичных полей.

accepted_if:anotherfield,value,...

Проверяемое поле должно иметь значение `"yes"`, `"on"`, `1` или `true`, если другое проверяемое поле равно указанному значению. Применяется для валидации принятия раздела «Условия использования» или аналогичных полей.

active_url

Проверяемое поле должно иметь допустимую запись A или AAAA в соответствии с функцией `dns_get_record` PHP. Имя хоста указанного URL извлекается с помощью PHP-функции `parse_url` перед передачей в `dns_get_record`.

after:*date*

Проверяемое поле должно иметь значение после указанной даты. Даты будут переданы в функцию `strtotime` PHP для преобразования в действительный экземпляр `DateTime`:

```
'start_date' => 'required|date|after:tomorrow'
```

Вместо передачи строки даты, которая будет проанализирована с помощью `strtotime`, вы можете указать другое поле для сравнения с датой:

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:*date*

Проверяемое поле должно иметь значение после указанной даты или равное ей. Для получения дополнительной информации см. правило after.

alpha

Проверяемое поле должно состоять полностью из букв.

alpha_dash

Проверяемое поле может содержать буквенно-цифровые символы, а также дефисы и подчеркивания.

alpha_num

Проверяемое поле должно состоять полностью из буквенно-цифровых символов.

array

Проверяемое поле должно быть массивом PHP.

Когда для правила `array` предоставляются дополнительные значения, каждый ключ во входном массиве должен присутствовать в списке значений, предоставленных правилу. В следующем примере ключ `admin` во входном массиве является недействительным, так как он не содержится в списке значений, предоставленных правилу `array`:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:username,locale',
]);
```

В общем, вы всегда должны указывать ключи массива, которые могут присутствовать в вашем массиве.

bail

Остановить дальнейшее применение правил валидации атрибута после первой неуспешной проверки.

В отличии от правила `bail`, которое прекращает дальнейшую валидацию только конкретного поля, метод `stopOnFirstFailure` сообщит валидатору, что он должен прекратить дальнейшую валидацию всех атрибутов при возникновении первой ошибке:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

before:*date*

Проверяемое поле должно быть значением, предшествующим указанной дате. Даты будут переданы в функцию PHP `strtotime` для преобразования в действительный экземпляр `DateTime`. Кроме того, как и в правиле `after`, имя другого проверяемого поля может быть указано в качестве значения `date`.

before_or_equal:*date*

Проверяемое поле должно иметь значение, предшествующее указанной дате или равное ей. Даты будут переданы в функцию PHP `strtotime` для преобразования в действительный экземпляр `DateTime`. Кроме того, как и в правиле `after`, имя другого проверяемого поля может быть указано в качестве значения `date`.

between:*min*,*max*

Проверяемое поле должно иметь размер между указанными *min* и *max*. Строки, числа, массивы и файлы оцениваются так же, как и в правиле `size`.

boolean

Проверяемое поле должно иметь возможность преобразования в логическое значение. Допустимые значения: `true`, `false`, `1`, `0`, `"1"` и `"0"`.

confirmed

Проверяемое поле должно иметь совпадающее поле `{field}_confirmation`. Например, если проверяемое поле – `password`, то поле `password_confirmation` также должно присутствовать во входящих данных.

current_password

Проверяемое поле должно соответствовать паролю аутентифицированного пользователя. Вы можете указать охранника аутентификации, используя первый параметр правила:

```
'password' => 'current_password:api'
```

date

Проверяемое поле должно быть действительной, не относительной датой в соответствии с функцией `strtotime` PHP.

date_equals:*date*

Проверяемое поле должно быть равно указанной дате. Даты будут переданы в функцию `strtotime` PHP для преобразования в действительный экземпляр `DateTime`.

date_format:*format*

Проверяемое поле должно соответствовать переданному *format*. При валидации поля следует использовать **либо** `date`, **либо** `date_format`, а не то и другое вместе. Это правило валидации поддерживает все форматы, поддерживаемые классом `DateTime` PHP.

declined

Проверяемое поле должно иметь значение `"no"`, `"off"`, `0` или `false`.

declined_if:anotherfield,value,...

Проверяемое поле должно иметь значение `"no"`, `"off"`, `0` или `false`, если другое проверяемое поле равно указанному значению.

different:*field*

Проверяемое поле должно иметь значение, отличное от *field*.

digits:*value*

Проверяемое целое число должно иметь точную длину *value*.

digits_between:*min*,*max*

Проверяемое целое число должно иметь длину между переданными *min* и *max*.

dimensions

Проверяемый файл должен быть изображением, отвечающим ограничениям размеров, указанным в параметрах правила:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Доступные ограничения: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.

Ограничение *ratio* должно быть представлено как ширина, разделенная на высоту. Это может быть указано дробью вроде `3/2` или числом с плавающей запятой, например `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Поскольку это правило требует нескольких аргументов, вы можете использовать метод `Rule::dimensions` для гибкости составления правила:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
    ],
]);
```

distinct

При валидации массивов проверяемое поле не должно иметь повторяющихся значений:

```
'foo.*.id' => 'distinct'
```

По умолчанию правило `distinct` использует гибкое сравнение переменных. Чтобы использовать жесткое сравнение, вы можете добавить параметр `strict` в определение правила валидации:

```
'foo.*.id' => 'distinct:strict'
```

Вы можете добавить `ignore_case` к аргументам правила валидации, чтобы правило игнорировало различия в использовании регистра букв:

```
'foo.*.id' => 'distinct:ignore_case'
```

email

Проверяемое поле должно быть отформатировано как адрес электронной почты. Это правило валидации использует пакет `egulias/email-validator` для проверки адреса электронной почты. По умолчанию применяется валидатор `RFCValidation`, но вы также можете применить другие стили валидации:

```
'email' => 'email:rfc,dns'
```

В приведенном выше примере будут применяться проверки `RFCValidation` и `DNSCheckValidation`. Вот полный список стилей проверки, которые вы можете применить:

- `rfc`: `RFCValidation`
- `strict`: `NoRFCWarningsValidation`
- `dns`: `DNSCheckValidation`
- `spoof`: `SpoofCheckValidation`
- `filter`: `FilterEmailValidation`

Валидатор `filter`, который использует функцию `filter_var` PHP, поставляется с Laravel и применялся по умолчанию до Laravel версии 5.8.

> {note} Валидаторы `dns` и `spoof` требуют расширения `intl` PHP.

ends_with:*foo,bar,...*

Проверяемое поле должно заканчиваться одним из указанных значений.

enum

Правило `Enum` – это правило на основе класса, которое определяет, имеет ли проверяемое поле допустимое значение перечисления. Правило `Enum` принимает имя перечисления как единственный аргумент конструктора:

```
use App\Enums\ServerStatus;
use Illuminate\Validation\Rules\Enum;

$request->validate([
    'status' => [new Enum(ServerStatus::class)],
]);
```

> {note} Перечисляемые типы доступны только в PHP 8.1+.

exclude

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`.

exclude_if:*anotherfield*,*value*

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если поле *anotherfield* равно *value*.

Если требуется сложная логика условного исключения, то вы можете использовать метод `Rule::excludeIf`. Этот метод принимает логическое значение или замыкание. При задании замыкания оно должно возвращать `true` или `false` для указания, следует ли исключить проверяемое поле:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::excludeIf(fn () => $request->user()->is_admin),
]);
```

exclude_unless:*anotherfield*,*value*

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если поле *anotherfield* не равно *value*. Если *value* равно `null` (т.е. `exclude_unless:name,null`), то проверяемое поле будет исключено, если поле сравнения не имеет значение `null` или поле сравнения отсутствует в данных запроса.

exclude_with:*anotherfield*

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если присутствует поле *anotherfield*.

exclude_without:*anotherfield*

Проверяемое поле будет исключено из данных запроса, возвращаемых методами `validate` и `validated`, если поле *anotherfield* отсутствует.

exists:*table*,*column*

Проверяемое поле должно существовать в указанной таблице базы данных.

Основы использования правила Exists

```
'state' => 'exists:states'
```

Если параметр `column` не указан, будет использоваться имя поля. Таким образом, в этом случае правило будет проверять, что таблица базы данных `states` содержит запись со значением столбца `state`, соответствующим значению атрибута `state` запроса.

Указание пользовательского имени столбца

Вы можете явно указать имя столбца базы данных, которое должно использоваться правилом валидации, поместив его после имени таблицы базы данных:

```
'state' => 'exists:states,abbreviation'
```

Иногда требуется указать конкретное соединение с базой данных, которое будет использоваться для запроса `exists`. Вы можете сделать это, добавив имя подключения к имени таблицы:

```
'email' => 'exists:connection.staff,email'
```

Вместо того, чтобы указывать имя таблицы напрямую, вы можете указать модель Eloquent, которая должна использоваться для определения имени таблицы:

```
'user_id' => 'exists:App\Models\User,id'
```

Если вы хотите использовать свой запрос, выполняемый правилом валидации, то вы можете использовать класс `Rule` для гибкости определения правила. В этом примере мы также укажем правила валидации в виде массива вместо использования символа `|` для их разделения:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            return $query->where('account_id', 1);
        }),
    ],
]);
```

Вы можете явно указать имя столбца базы данных, которое должно использоваться правилом `exists`, сгенерированным методом `Rule::exists`, указав имя столбца в качестве второго аргумента метода `exists`:

```
'state' => Rule::exists('states', 'abbreviation'),
```

file

Проверяемое поле должно быть успешно загруженным на сервер файлом.

filled

Проверяемое поле не должно быть пустым, если оно присутствует.

gt:*field*

Проверяемое поле должно быть больше указанного *field*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

gte:*field*

Проверяемое поле должно быть больше или равно указанному *field*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

image

Проверяемый файл должен быть изображением (jpg, jpeg, png, bmp, gif, svg или webp).

in:*foo*,*bar*,...

Проверяемое поле должно быть включено в указанный список значений. Поскольку это правило часто требует, чтобы вы «объединяли» массив, то метод `Rule::in` можно использовать для гибкого построения правила:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

Когда правило `in` комбинируется с правилом `array`, тогда каждое значение во входном массиве должно присутствовать в списке значений, предоставленных правилу `in`. В следующем примере код аэропорта `LAS` во входном массиве является недействительным, так как он не содержится в списке аэропортов, предоставленном правилу `in`:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$input = [
    'airports' => ['NYC', 'LAS'],
];

Validator::make($input, [
    'airports' => [
        'required',
        'array',
    ],
    'airports.*' => Rule::in(['NYC', 'LIT']),
]);
```

in_array:*anotherfield*.*

Проверяемое поле должно существовать в значениях *anotherfield*.

integer

Проверяемое поле должно быть целым числом.

> {note} Это правило валидации не проверяет, что значение поля относится к типу переменной `integer`, а только что значение поля относится к типу, принятому правилом `FILTER_VALIDATE_INT` PHP. Если вам нужно проверить значение поля в качестве числа, используйте это правило в сочетании с правилом валидации `numeric`.

ip

Проверяемое поле должно быть IP-адресом.

ipv4

Проверяемое поле должно быть адресом IPv4.

ipv6

Проверяемое поле должно быть адресом IPv6.

json

Проверяемое поле должно быть допустимой строкой JSON.

lt:*field*

Проверяемое поле должно быть меньше переданного *field*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

lte:*field*

Проверяемое поле должно быть меньше или равно переданному *field*. Два поля должны быть одного типа. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

mac_address

Проверяемое поле должно быть MAC-адресом.

max:*value*

Проверяемое поле должно быть меньше или равно максимальному *value*. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

mimetypes:*text/plain*,...

Проверяемый файл должен соответствовать одному из указанных MIME-типов:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

Чтобы определить MIME-тип загруженного файла, содержимое файла будет прочитано, и фреймворк попытается угадать MIME-тип, который может отличаться от типа, предоставленного клиентом.

mimes:*foo*,*bar*,...

Проверяемый файл должен иметь MIME-тип, соответствующий одному из перечисленных расширений.

Основы использования правила MIME

```
'photo' => 'mimes:jpg,bmp,png'
```

Несмотря на то, что вам нужно только указать расширения, это правило фактически проверяет MIME-тип файла, читая содержимое файла и угадывая его MIME-тип. Полный список типов MIME и соответствующих им расширений можно найти по следующему адресу:

https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:*value*

Проверяемое поле должно иметь минимальное значение *value*. Строки, числа, массивы и файлы оцениваются с использованием тех же соглашений, что и в правиле `size`.

multiple_of:*value*

Проверяемое поле должно быть кратным *value*.

> {note} Для использования правила `multiple_of` требуется расширение `bcmath` PHP.

not_in:*foo*,*bar*,...

Проверяемое поле не должно быть включено в переданный список значений. Метод `Rule::notIn` используется для гибкого построения правила:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

not_regex:*pattern*

Проверяемое поле не должно соответствовать переданному регулярному выражению.

Внутренне это правило использует функцию `preg_match` PHP. Указанный шаблон должен подчиняться тому же форматированию, требуемому `preg_match`, и, следовательно, также включать допустимые разделители. Например: `'email' => 'not_regex:/^.+$/i'`.

> {note} При использовании шаблонов `regex` / `not_regex` может потребоваться указать ваши правила валидации с использованием массива вместо использования разделителей `|`, особенно если регулярное выражение содержит символ `|`.

nullable

Проверяемое поле может быть `null`.

numeric

Проверяемое поле должно быть числовым.

password

Проверяемое поле должно соответствовать паролю аутентифицированного пользователя.

> {note} Это правило было переименовано в `current_password` с его дальнейшим удалением в Laravel 9. Вместо этого используйте правило Current Password.

present

Проверяемое поле должно присутствовать во входных данных, но может быть пустым.

prohibited

Проверяемое поле должно быть пустым или отсутствовать.

prohibited_if:*anotherfield*,*value*,...

Проверяемое поле должно быть пустым или отсутствовать, если поле *anotherfield* равно любому *value*.

Если требуется сложная логика условного запрета присутствия, то вы можете использовать метод `Rule::prohibitedIf`. Этот метод принимает логическое значение или замыкание. При задании замыкания оно должно возвращать `true` или `false` для указания, следует ли запретить присутствие проверяемого поля:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::prohibitedIf(fn () => $request->user()->is_admin),
]);
```

prohibited_unless:*anotherfield*,*value*,...

Проверяемое поле должно быть пустым или отсутствовать, если поле *anotherfield* не равно какому-либо *value*.

prohibits:*anotherfield*,...

Если проверяемое поле присутствует, никакие поля в *anotherfield* не могут присутствовать, даже если они пустые.

regex:*pattern*

Проверяемое поле должно соответствовать переданному регулярному выражению.

Внутренне это правило использует функцию `preg_match` PHP. Указанный шаблон должен подчиняться тому же форматированию, требуемому `preg_match`, и, следовательно, также включать допустимые разделители. Например: `'email' => 'regex:/^.+@.+$/i'`.

> {note} При использовании шаблонов `regex` / `not_regex` может потребоваться указать ваши правила валидации с использованием массива вместо использования разделителей `|`, особенно если регулярное выражение содержит символ `|`.

required

Проверяемое поле должно присутствовать во входных данных и не быть пустым. Поле считается «пустым», если выполняется одно из следующих условий:

- Значение поля равно `null`.
- Значение поля – пустая строка.

- Значение поля представляет собой пустой массив или пустой объект, реализующий интерфейс `Countable`.
- Значение поля – загружаемый файл, но без пути.

required_if:*anotherfield*,*value*,...

Проверяемое поле должно присутствовать и не быть пустым, если поле *anotherfield* равно любому *value*.

Если вы хотите создать более сложное условие для правила `required_if`, вы можете использовать метод `Rule::requiredIf`. Этот метод принимает логическое значение или замыкание. При выполнении замыкания оно должно возвращать `true` или `false`, чтобы указать, обязательно ли проверяемое поле:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf(fn () => $request->user()->is_admin),
]);
```

required_unless:*anotherfield*,*value*,...

Проверяемое поле должно присутствовать и не быть пустым, если поле *anotherfield* не равно какому-либо *value*. Это также означает, что в данных запроса должно присутствовать *anotherfield*, если *value* не имеет значения `null`. Если *value* равно `null` (т.е. `required_unless:name,null`), то проверяемое поле будет обязательным, если поле сравнения не равно `null` или поле сравнения отсутствует в данных запроса.

required_with:*foo*,*bar*,...

Проверяемое поле должно присутствовать и не быть пустым, *только если* любое из других указанных полей присутствует и не является пустым.

required_with_all:*foo*,*bar*,...

Проверяемое поле должно присутствовать и не быть пустым, *только если* все другие указанные поля присутствуют и не являются пустыми.

required_without:*foo*,*bar*,...

Проверяемое поле должно присутствовать и не быть пустым, *только когда* любое из других указанных полей является пустым или отсутствует.

required_without_all:*foo*,*bar*,...

Проверяемое поле должно присутствовать и не быть пустым, *только когда* все другие указанные поля являются пустыми или отсутствуют.

required_array_keys:*foo,bar,...*

Проверяемое поле должно быть массивом и содержать как минимум указанные ключи.

same:*field*

Переданное *field* должно соответствовать проверяемому полю.

size:*value*

Проверяемое поле должно иметь размер, соответствующий переданному *value*. Для строковых данных *value* соответствует количеству символов. Для числовых данных *value* соответствует переданному целочисленному значению (атрибут также должен иметь правило `numeric` или `integer`). Для массива *size* соответствует `count` массива. Для файлов *size* соответствует размеру файла в килобайтах. Давайте посмотрим на несколько примеров:

```
// Проверяем, что строка содержит ровно 12 символов ...
'title' => 'size:12';

// Проверяем, что передано целое число, равно 10 ...
'seats' => 'integer|size:10';

// Проверяем, что в массиве ровно 5 элементов ...
'tags' => 'array|size:5';

// Проверяем, что размер загружаемого файла составляет ровно 512 килобайт ...
'image' => 'file|size:512';
```

starts_with:*foo,bar,...*

Проверяемое поле должно начинаться с одного из указанных значений.

string

Проверяемое поле должно быть строкой. Если вы хотите, чтобы поле также могло быть `null`, вы должны назначить этому полю правило `nullable`.

timezone

Проверяемое поле должно быть допустимым идентификатором часового пояса в соответствии с функцией `timezone_identifiers_list` PHP.

unique:*table,column*

Проверяемое поле не должно существовать в указанной таблице базы данных.

Указание пользовательского имени таблицы / имени столбца:

Вместо того, чтобы указывать имя таблицы напрямую, вы можете указать модель Eloquent, которая должна использоваться для определения имени таблицы:

```
'email' => 'unique:App\Models\User,email_address'
```

Параметр `column` используется для указания соответствующего столбца базы данных поля. Если опция `column` не указана, будет использоваться имя проверяемого поля.

```
'email' => 'unique:users,email_address'
```

Указание пользовательского соединения базы данных

Иногда требуется указать конкретное соединение для запросов к базе данных, выполняемых валидатором. Для этого вы можете добавить имя подключения к имени таблицы:

```
'email' => 'unique:connection.users,email_address'
```

Принудительное игнорирование правилом Unique конкретного идентификатора:

Иногда вы можете проигнорировать конкретный идентификатор во время валидации `unique`. Например, рассмотрим страницу «Обновления профиля», которая включает имя пользователя, адрес электронной почты и местоположение. Вероятно, вы захотите убедиться, что адрес электронной почты уникален. Однако, если пользователь изменяет только поле имени, а не поле электронной почты, то вы не захотите, чтобы выдавалась ошибка валидация, поскольку пользователь уже является владельцем рассматриваемого адреса электронной почты.

Чтобы указать валидатору игнорировать идентификатор пользователя, мы воспользуемся классом `Rule` для гибкого определения правила. В этом примере мы также укажем правила валидации в виде массива вместо использования символа `|` для их разделения:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

> {note} Вы никогда не должны передавать какое-либо введенное пользователем значение из запроса в метод `ignore`. Вместо этого вы должны передавать только сгенерированный системой уникальный идентификатор, такой как автоинкрементный идентификатор или UUID экземпляра модели Eloquent. В противном случае ваше приложение будет уязвимо для атаки с использованием SQL-инъекции.

Вместо того, чтобы передавать значение ключа модели методу `ignore`, вы также можете передать весь экземпляр модели. Laravel автоматически извлечет ключ из модели:

```
Rule::unique('users')->ignore($user)
```

Если ваша таблица использует имя столбца с первичным ключом, отличное от `id`, то вы можете указать имя столбца при вызове метода `ignore`:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

По умолчанию правило `unique` проверяет уникальность столбца, совпадающего с именем проверяемого атрибута. Однако вы можете передать другое имя столбца в качестве второго аргумента метода `unique`:

```
Rule::unique('users', 'email_address')->ignore($user->id),
```

Добавление дополнительных выражений Where:

Вы можете указать дополнительные условия запроса, изменив запрос с помощью метода `where`. Например, давайте добавим условие запроса, которое ограничивает область запроса только поиском записями, у которых значение столбца `account_id` равно `1`:

```
'email' => Rule::unique('users')->where(fn ($query) => $query->where('account_id', 1))
```

url

Проверяемое поле должно быть действительным URL.

uuid

Проверяемое поле должно быть действительным универсальным уникальным идентификатором (UUID) RFC 4122 (версии 1, 3, 4 или 5).

Условное добавление правил

Пропуск валидации при определенных значениях полей

По желанию можно не проверять конкретное поле, если другое поле имеет указанное значение. Вы можете сделать это, используя правило валидации `exclude_if`. В этом примере поля `appointment_date` и `doctor_name` не будут проверяться, если поле `has_appointment` имеет значение `false`:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [
```

```
    'has_appointment' => 'required|boolean',
    'appointment_date' => 'exclude_if:has_appointment,false|required|date',
    'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);
```

В качестве альтернативы вы можете использовать правило `exclude_unless`, чтобы не проверять конкретное поле, если другое поле не имеет указанного значения:

```
$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' => 'exclude_unless:has_appointment,true|required|date',
    'doctor_name' => 'exclude_unless:has_appointment,true|required|string',
]);
```

Валидация при условии наличия

По желанию можно выполнить валидацию поля, **только** если это поле присутствует в проверяемых данных. Чтобы этого добиться, добавьте правило `sometimes` в свой список правил:

```
$validator = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

В приведенном выше примере поле `email` будет проверено, только если оно присутствует в массиве `$request->all()`.

> {tip} Если вы пытаетесь проверить поле, которое всегда должно присутствовать, но может быть пустым, ознакомьтесь с этим примечанием о необязательных полях.

Комплексная условная проверка

Иногда вы можете добавить правила валидации, основанные на более сложной условной логике. Например, вы можете потребовать обязательного присутствия переданного поля только в том случае, если другое поле имеет значение больше 100. Или вам может потребоваться, чтобы два поля имели указанное значение только при наличии другого поля. Добавление этих правил валидации не должно вызывать затруднений. Сначала создайте экземпляр `Validator` со своими *статическими правилами*, которые будут неизменными:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Предположим, что наше веб-приложение предназначено для коллекционеров игр. Если коллекционер игр регистрируется в нашем приложении и у него есть более 100 игр, мы хотим, чтобы он объяснил, почему у него так много игр. Например, возможно, они владеют магазином по перепродаже игр или, может быть, им просто нравится коллекционировать игры. Чтобы условно добавить это требование, мы можем использовать метод `sometimes` экземпляра `Validator`:

```
$validator->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

Первый аргумент, переданный методу `sometimes` – это имя поля, которое мы условно проверяем. Второй аргумент – это список правил, которые мы хотим добавить. Если замыкание, переданное в качестве третьего аргумента, возвращает `true`, то правила будут добавлены. Этот метод упрощает создание сложных условных проверок. Вы даже можете добавить условные проверки сразу для нескольких полей:

```
$validator->sometimes(['reason', 'cost'], 'required', function ($input) {
    return $input->games >= 100;
});
```

> {tip} Параметр `$input`, переданный вашему замыканию, будет экземпляром `Illuminate\Support\Fluent` и может использоваться при валидации для доступа к вашим входящим данным и файлам запроса.

Комплексная условная валидация массива

Иногда требуется проверить поле на основе другого поля в том же вложенном массиве, индекс которого вам неизвестен. В этих ситуациях вы можете использовать второй аргумент вашего замыкания, который будет текущим отдельным элементом в проверяемом массиве:

```
$input = [
    'channels' => [
        [
            'type' => 'email',
            'address' => 'abigail@example.com',
        ],
        [
            'type' => 'url',
            'address' => 'https://example.com',
        ],
    ],
];

$validator->sometimes('channels.*.address', 'email', function ($input, $item) {
    return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function ($input, $item) {
```

```
    return $item->type !== 'email';
});
```

Подобно параметру `$input`, переданному в замыкание, параметр `$item` будет являться экземпляром `Illuminate\Support\Fluent`, если атрибут данных является массивом; в противном случае – это строка.

Валидация массивов

Как обсуждалось в правилах валидации массива, правило `array` принимает список разрешенных ключей массива. Если в массиве присутствуют какие-либо дополнительные ключи, проверка не будет пройдена:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:username,locale',
]);
```

В общем, вы всегда должны указывать ключи массива, которые могут присутствовать в вашем массиве. В противном случае методы валидатора `validate` и `validated` вернут все провалидированные данные, включая массив и все его ключи, даже если эти ключи не были провалидированы другими правилами валидации вложенных массивов.

Валидация вложенных массивов

Проверка полей ввода формы на основе массива не должна быть проблемой. Вы можете использовать «точечную нотацию» для валидации атрибутов в массиве. Например, если входящий HTTP-запрос содержит поле `photos[profile]`, вы можете проверить его следующим образом:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

Вы также можете проверить каждый элемент массива. Например, чтобы убедиться, что каждое электронное письмо в переданном поле ввода массива уникально, вы можете сделать следующее:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Точно так же вы можете использовать символ `*` при указании пользовательских сообщений в ваших языковых файлах, что упрощает использование одного сообщения валидации для полей на основе массива:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique email address',
    ]
],
```

Доступ к данным вложенного массива

Иногда требуется получить доступ к значению переданного вложенного элемента массива при назначении правил валидации для атрибута. Вы можете сделать это, используя метод `Rule::forEach`. Метод `forEach` принимает замыкание, которое будет вызываться для каждой итерации проверяемого атрибута массива, и будет получать значение атрибута и явное, полностью развернутое имя атрибута. Замыкание должно возвращать массив правил предназначенных элементу массива:

```
use App\Rules\HasPermission;
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$validator = Validator::make($request->all(), [
    'companies.*.id' => Rule::forEach(function ($value, $attribute) {
        return [
            Rule::exists(Company::class, 'id'),
            new HasPermission('manage-company', $value),
        ];
    }),
]);
```

Индексы и позиции сообщений об ошибках

При валидации массивов вы можете указать индекс или положение определенного элемента, который не прошел проверку, в сообщении об ошибке, отображаемом вашим приложением. Для этого вы можете включить заполнители `:index` и `:position` в свое собственное сообщение об ошибке:

```
use Illuminate\Support\Facades\Validator;

$input = [
```

```
    'photos' => [
        [
            'name' => 'BeachVacation.jpg',
            'description' => 'A photo of my beach vacation!',
        ],
        [
            'name' => 'GrandCanyon.jpg',
            'description' => '',
        ],
    ],
];

Validator::validate($input, [
    'photos.*.description' => 'required',
], [
    'photos.*.description.required' => 'Please describe photo #:position.',
]);
```

В приведенном выше примере валидация завершится ошибкой, и пользователю будет представлена следующая ошибка: "*Please describe photo #2.*"

Валидация паролей

Чтобы гарантировать, что пароли имеют адекватный уровень сложности, вы можете использовать класс правила `Password` Laravel:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\Password;

$validator = Validator::make($request->all(), [
    'password' => ['required', 'confirmed', Password::min(8)],
]);
```

Класс правила `Password` позволяет вам легко форсировать требования к сложности пароля для вашего приложения, например указать, что для паролей требуется хотя бы одна буква, цифра, символ или символы со смешанным регистром:

```
// Требуется не менее 8 символов ...
Password::min(8)

// Требуется хотя бы одна буква ...
Password::min(8)->letters()

// Требуется хотя бы одна заглавная и одна строчная буквы ...
Password::min(8)->mixedCase()

// Требуется хотя бы одна цифра ...
Password::min(8)->numbers()

// Требуется хотя бы один символ ...
Password::min(8)->symbols()
```

Кроме того, вы можете гарантировать, что пароль не был скомпрометирован при утечке данных, используя метод `uncompromised`:

```
Password::min(8)->uncompromised()
```

Внутренне объект правила `Password` использует модель k-Anonymity, чтобы определить, не произошла ли утечка пароля через сервис haveibeenpwned.com без ущерба для конфиденциальности или безопасности пользователя.

По умолчанию, если пароль фигурирует хотя бы один раз в утечке данных, он будет считаться скомпрометированным. Вы можете изменить этот критерий, используя первый аргумент метода `uncompromised`:

```
// Убедиться, что пароль появляется менее 3 раз в утечке данных ...
Password::min(8)->uncompromised(3);
```

Вы можете связать все методы в упомянутых выше примерах:

```
Password::min(8)
    ->letters()
    ->mixedCase()
    ->numbers()
    ->symbols()
    ->uncompromised()
```

Определение правил валидации паролей по умолчанию

Возможно, вам будет удобно указать правила валидации паролей по умолчанию в одном месте вашего приложения. Вы можете легко сделать это, используя метод `Password::defaults`, который принимает замыкание. Замыкание, переданное методу `defaults`, должно вернуть конфигурацию правила пароля по умолчанию. Обычно вызов метода `defaults` осуществляется в методе `boot` одного из поставщиков служб вашего приложения:

```
use Illuminate\Validation\Rules\Password;

/**
 * Загрузка любых служб приложения.
 *
 * @return void
 */
public function boot()
{
    Password::defaults(function () {
        $rule = Password::min(8);

        return $this->app->isProduction()
```

```
                    ? $rule->mixedCase()->uncompromised()
                    : $rule;
    });
}
```

Затем, когда вы хотите применить правила по умолчанию к конкретному паролю, проходящему валидацию, вы можете вызвать метод `defaults` без аргументов:

```
'password' => ['required', Password::defaults()],
```

По желанию можно добавить дополнительные правила валидации к правилам валидации пароля по умолчанию. Для этого вы можете использовать метод `rules`:

```
use App\Rules\ZxcvbnRule;

Password::defaults(function () {
    $rule = Password::min(8)->rules([new ZxcvbnRule]);

    // ...
});
```

Пользовательские правила валидации

Использование класса Rule

Laravel предлагает множество полезных правил валидации; однако вы можете указать свои собственные. Один из методов регистрации собственных правил валидации – использование объектов правил. Чтобы сгенерировать новый объект правила, вы можете использовать команду `make:rule` Artisan. Давайте воспользуемся этой командой, чтобы сгенерировать правило, которое проверяет, что строка состоит из прописных букв. Laravel поместит новый класс правила в каталог `app/Rules` вашего приложения. Если этот каталог не существует в вашем приложении, то Laravel предварительно создаст его, когда вы выполните команду Artisan для создания своего правила:

```
php artisan make:rule Uppercase
```

Как только правило создано, мы готовы определить его поведение. Объект правила содержит два метода: `passes` и `message`. Метод `passes` получает значение и имя атрибута и должен возвращать `true` или `false` в зависимости от того, является ли значение атрибута допустимым или нет. Метод `message` должен возвращать сообщение об ошибке валидации, которое следует использовать, если проверка оказалась не успешной:

```
<?php

namespace App\Rules;
```

```
use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule
{
    /**
     * Определить, пройдено ли правило валидации.
     *
     * @param  string  $attribute
     * @param  mixed  $value
     * @return bool
     */
    public function passes($attribute, $value)
    {
        return strtoupper($value) === $value;
    }

    /**
     * Получить сообщение об ошибке валидации.
     *
     * @return string
     */
    public function message()
    {
        return 'The :attribute must be uppercase.';
    }
}
```

Вы можете вызвать помощник `trans` в методе `message`, если хотите вернуть сообщение об ошибке из ваших файлов перевода:

```
/**
 * Получить сообщение об ошибке валидации.
 *
 * @return string
 */
public function message()
{
    return trans('validation.uppercase');
}
```

После определения правила вы можете отправить его валидатору, передав экземпляр объекта правила с другими вашими правилами валидации:

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
]);
```

Доступ к дополнительным данным

Если вашему пользовательскому классу правил валидации требуется доступ к другим валидируемым данным, то ваш класс правил может реализовать интерфейс `Illuminate\Contracts\Validation\DataAwareRule`. Этот интерфейс требует, чтобы ваш класс определял метод `setData`. Этот метод будет автоматически вызван Laravel (до того, как начнется валидация) со всеми проверяемыми данными:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\DataAwareRule;

class Uppercase implements Rule, DataAwareRule
{
    /**
     * Все валидируемые данные.
     *
     * @var array
     */
    protected $data = [];

    // ...

    /**
     * Установить валидируемые данные.
     *
     * @param  array  $data
     * @return $this
     */
    public function setData($data)
    {
        $this->data = $data;

        return $this;
    }
}
```

Или, если вашему правилу валидации требуется доступ к экземпляру текущего валидатора, то вы можете реализовать интерфейс `ValidatorAwareRule`:

```
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;

class Uppercase implements Rule, ValidatorAwareRule
{
    /**
     * Экземпляр валидатора.
     *
```

```
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...

    /**
     * Установить текущий валидатор.
     *
     * @param  \Illuminate\Validation\Validator  $validator
     * @return $this
     */
    public function setValidator($validator)
    {
        $this->validator = $validator;

        return $this;
    }
}
```

Использование замыканий

Если вам нужна функциональность собственного правила только один раз во всем приложении, то вы можете использовать анонимное правило вместо объекта правила. Анонимное правило получит имя атрибута, значение атрибута и замыкание `$fail`, которое должно быть вызвано при не успешной проверки:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function ($attribute, $value, $fail) {
            if ($value === 'foo') {
                $fail('The '.$attribute.' is invalid.');
            }
        },
    ],
]);
```

Неявные правила

По умолчанию, правила валидации, включая созданные вами, не применяются, если проверяемый атрибут отсутствует или содержит пустую строку. Например, правило `unique` не будет выполнено для пустой строки:

```
use Illuminate\Support\Facades\Validator;

$rules = ['name' => 'unique:users,name'];
```

```
$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true
```

Чтобы ваше правило было применено, даже если атрибут пуст, то правило должно подразумевать, что атрибут является обязательным. Чтобы создать «неявное» правило, реализуйте интерфейс `Illuminate\Contracts\Validation\ImplicitRule` . Это «маркерный интерфейс» для валидатора; следовательно, он не содержит никаких дополнительных методов, которые вам нужно реализовать, помимо методов, требуемых типичным интерфейсом `Rule` .

Чтобы сгенерировать новый объект неявного правила, вы можете использовать команду `make:rule` Artisan с флагом `--implicit` :

```
php artisan make:rule Uppercase --implicit
```

> {note} «Неявное» правило только *подразумевает*, что атрибут является обязательным к валидации. В действительности, решать только вам, будет ли пустой или отсутствующий атрибут считаться невалидным.